(4)

AD-A227 632 T DOCUMENTATION PAGE

DTIC FILE COPY

Form Approved
OMB No. 0704-0188

DTIC
ELECTE
OCT 17 1990
S B D

1a. REPORT SECURITY CLASSIFICATION Unclassified/Unlimited	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) 5 24088	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Rensselaer Polytechnic Institute 6b. OFFICE SYMBOL (If applicable) 3A707	7a. NAME OF MONITORING ORGANIZATION SCD-C Office of Naval Research Resident Representative N62927
6c. ADDRESS (City, State, and ZIP Code) Office of Contracts & Grants Troy, New York 12181	7b. ADDRESS (City, State, and ZIP Code) 33 Third Avenue - Lower Level New York, NY 10002-9998
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Dept. of Navy Office of the Chief of Naval Research 8b. OFFICE SYMBOL (If applicable) N00014	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 800 No. Quincy Street Arlington, VA 22217-5000	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)
Generating Parallel and Real-Time Systems From Equational Programming Language Specifications

12. PERSONAL AUTHOR(S)
Boleslaw K. Szymanski

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM 86/07/01 TO 90/09/30	1990/09/30	8 pages

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Parallel Programming, Configurator, Real-Time Programming, Program Integration, Data Dependence, Annotations

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Equational Programming Language, abbreviated EPL, has been designed and implemented at Rensselaer Polytechnic Institute to increase productivity of programmers in the area of real-time and parallel programming. EPL is a simple non-strict functional language with type inference. The language is defined in terms of just a few constructs: generalized arrays and subscripts for data structures, recurrent equations for data value definitions, ports for process interactions and virtual processors for execution directives. Yet, its powerful compiler can generate object code for a variety of parallel and distributed architectures. The compilation is based on conditional data dependence analysis and data attribute propagation. The interplay between the user supplied annotations and compiler transformation has also been investigated. The software tools for integration of EPL programs into a parallel computation have been developed.

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ OTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

Final Report

for the ONR contract N00014-86-K-0442

by

Boleslaw K. Szymanski
Principal Investigator

Difficulties arising in adapting and using traditional high-level languages to program parallel and distributed computations have increased interest in alternative programming paradigms, such as dataflow, assertive, reduction, logic, etc. In many of these paradigms proposed languages belong to the larger class of functional languages.

A functional program defines a set of functions. An execution of a functional program can be viewed as an application of a function to a set of values of its parameters. Since there is no notion of the program state, side effects of conventional programming languages are completely absent in the functional programs. Consequently, concurrent execution of multiple functions is permitted. The usefulness of the functional languages in parallel programming can be further advanced if the referential transparency is supported by allowing only definitions and not assignments in the program. The referential transparency simplifies compile-time program transformation, data dependence analysis and parallelization.

In one of the novel approaches to parallel programming, called assertive paradigm, computations are specified as sets of assertions about properties of the solution, and not as sequences of procedural steps. Procedural solutions are automatically generated from the assertive description. Programmers are not involved in the detailed implementation, as efficiency and correctness are assured by the underlying language translator.

Depending on the type of assertions that are used as a basis for a notation, different languages for assertive programming have been proposed. Perhaps the best known is logic programming with the Prolog language as its prime example. In Prolog, assertions are expressed as Horn clauses. Automatic inference of new facts from the given rules and known facts makes it a convenient programming tool for artificial intelligence and expert systems. However, in applications in which numerical computations are involved, Prolog usefulness is questionable because of the inconvenience of expressing numerical algorithms in that language.

Another notation for assertive programming was proposed in equational languages, where assertions are expressed as algebraic equations. Programs written in equational languages are concise, free from implementation details, and easily amenable to verification and parallel processing. Those programs, however, require a sophisticated translator to generate efficient object code. It is necessary to use global analysis

and heuristic program transformations to achieve a quality translation. The envisaged role of the computer is not to execute, step by step, prescribed operations, like in procedural programming, but to find such values of unknown variables, that all stated assertions become true.

A modern scientific and engineering computation language should satisfy several additional postulates arising from the plethora of architectures on which such computations can be executed. The most important postulate is to separate the issue of execution from the meaning of the computation. In other words, the language should enable the programmer to separate 'what' from 'how'; the description of the meaning of the computation should be separated from the statements (if any) directing the compiler in translating the computation for execution on any particular architecture. Such a feature would also contribute to rapid prototyping and increased portability of the code.

In defining any large-scale computation, proper facilities for problem decompositions are of the utmost importance. The description of the computation in each of the decomposed subtasks should be separate from the description of the interactions between those subtasks. In conventional programming, the first description roughly corresponds to the programming-in-the-small, and the second one to the programming-in-the-large. The language for large-scale scientific and engineering programming should provide means for keeping these descriptions independent.

An assignment statement, the cornerstone of any conventional programming language, is the main source of difficulty in parallel programming, since it changes the value of the variable on the left-hand side of the assignment. Thus, a value of any variable at a certain point of the program execution is defined by the last executed assignment statement for this variable. Consequently, execution of any procedure that uses global variables is affected not only by the values of its parameters but also by the assignments to these global variables. Likewise, effects of the procedure execution may include changes to the values of the global variables that have assignment statements in the procedure body. If there are parallel execution paths, each containing assignment statements for a variable, then the final value of this variable may depend on a relative speed with which those paths are executed. To avoid such side effects of the assignment statement, many parallel functional languages enforce the single assignment rule which states that each variable can have only one value and prior to the assignment of a value, the variable is undefined.

The single assignment rule leads to declaring and operating on structures that have 'excessive' dimensionality. After all, a variable which would be merely reassigned in a traditional language, in the presence of a single assignment rule, has to be viewed as a vector of values and each reassignment has to be indexed by a different subscript. However, the multidimensional view of such a variable is logical only. The optimizing compiler should easily be able to eliminate such an excessive dimension in the variable implementation. If there are several candidate dimensions for elimination, the compiler can select an elimination by analyzing all data dependences in a computation. Thus, it is reasonable to expect that such a selection will be at least as good as the selection made by the programmer who typically bases the decision more on

n For
?A&I
3
:ced
:ation
ution/
bility Codes
ail and/or
Special
Dist
A-1

intuition and meaning of the variable than on the implementation efficiency.

Finally, to enable a language compiler to explore parallelism at its lowest level, the language should also provide operators which can be applied to components of the arguments in a dataflow fashion, i.e., in the order that those components become available, and not in the order of their declaration.

Equational languages are naturally suited to mathematical modeling. They are convenient to describe computations that involve solving systems of linear equations that may arise directly (as, e.g., in econometric modeling or as the result of a discrete approximation of a system of differential equations. Various numerical aspects of the solution, such as the applied method, initial values or convergence criteria, can be either generated automatically by default or may be provided by the programmer. Equational languages have also been proven to be an effective tool for describing general computational tasks.

Equational Programming Language, abbreviated EPL, is a simple non-strict functional language with type inference designed for programming parallel and distributed computation. The language is defined in terms of just a few constructs: generalized arrays and subscripts for data structures, recurrent equations for data value definitions, ports for process interactions and virtual processors for execution directives. An EPL program consists of data declarations and annotated conditional equations. Equations are defined over multidimensional jagged-edge arrays and may be annotated by virtual processors on which they are executed. Data declarations are annotated by the record and port designators which are used to identify interfaces with an external environment and other programs.

In addition to programs, the EPL user can define configurations which describe interconnections between ports of different processes. Configurations allow the programmer to reuse the same EPL programs in different computations. They also facilitate computation decomposition. A port creates a fair merge of its input sequences and hence enables an easy expression of non-determinism without changing the functional character of a program definition.

In addition to single-valued data structures, EPL programs contain subscripts that assume a range of integers as their values. Subscripts give EPL a dual flavor. In the definitional view they may be treated as universal quantifiers and equations are then viewed as logical predicates. In the operational view they can be seen as loop control variables, and each equation then is seen as a statement nested in loops implied by its subscripts. A more detailed description of the language may be found elsewhere.

The basic techniques used in the compilation of EPL programs are data dependence analysis and data attributes propagation. In a single program, the dependencies are represented in a compact form by the conditional array graph. This graph associates each dependence with its attributes, such as the distance between dependent elements, conditions under which dependence holds, the subscripts associated with it, etc. Both explicit data dependencies (defined by the usage of one data structure in an equation defining the other) and implicit data dependencies (implied, for example, by the

sequentiality of the reading of incoming messages) are represented as various kinds of edges in the conditional array graph. Each node of this graph contains information about the represented entity, such as the number and ranges of its dimensions, its type and class, and conditions guarding its definitions.

The correctness of the program is checked by verifying the consistency of the different attributes of data structures and data dependencies. To accomplish this, the EPL compiler propagates data and dependence attributes along the edges of the graph.

A similar dependence graph is also created for a configuration. It shows the data dependences among the processes of the computation and is used in scheduling processes and mapping them onto the processors.

The extent of transformation required to generate the object code depends on the architecture at hand. Similarly as for LUCID that is presented in the second chapter, EPL programs can be almost directly executed on a specialized tagged dataflow architecture that consists of the following functional elements:

token memory: A memory in which each tagged value (subscripted variable) is stored after it has been read in or evaluated.

matching unit: A unit that releases an equation instance for execution when all data values needed for that equation evaluation are present in the token memory.

executing unit: One of a number of arithmetic units able to evaluate EPL operators.

The conditional array graph defines the number of copies of a value needed in the token memory for each evaluated subscripted variable. One copy is needed for each edge outgoing from the node representing the corresponding variable. Each process may have a separate dataflow machine assigned for its execution and these machines have to be connected to exchange data through process ports. Alternatively, one dataflow machine may be allocated to the entire computation, and then ports would merely define equivalences of variables from different processes. With a sufficient number of arithmetic units, the dataflow implementation can provide the highest parallelism. However, eager scheduling of EPL computations can easily lead to an excessive demand for the token memory.

In the well-known Flynn's classification of parallel computational models, the von Neumann model is characterized by a Single stream of Instructions controlling a Single stream of Data (SISD). To achieve parallelism, Multiple Data streams have been introduced creating (SIMD) model. Further extension is to add Multiple Instruction streams and this extension leads to (MIMD) architectures. The last category can be conveniently split on the basis of a data access mechanism into shared and distributed memory architectures. In the shared-memory architectures processors have an equal access to one global memory. In the distributed-memory architectures, each processor has a direct access to its local memory and indirect access to the memory of other processors. The indirect access is typically supported through message passing mechanism that enables processors to communicate with each other.

For a SIMD machine, such as the Connection Machine, the major task of the EPL translation is to identify an EPL subscript that will index individual processors (i.e., a subscript that defines a domain of the computation). Equations indexed by the domain subscript will be executed in parallel on different processors. The selection of the domain subscript is influenced by the fact that a reference to an indexing expression different than a domain subscript implies communication of data from another processor.

Even more involving is the translation for Multiple Instruction Multiple Data (MIMID) machines. For shared-memory architectures in this class, only the placement of equations is an issue. The ports can be easily and efficiently implemented as blocks of shared-memory. The efficient translation requires strong memory optimization to counterweight the effects of 'excessive' dimensions present due to the single assignment rule of the language.

For distributed-memory machines the additional difficulty arises from data placement. In the current implementation of the EPL code generator for the Intel hypercube, it is assumed that data are distributed together with the equations that define them. This assumption makes the optimal allocation of equations to processors a more complex task.

The more detailed discussion and documentation of the work done in this project is presented in the cited below nine papers and six Technical Reports report prepared in connection with the research performed under the reported contract.

The general description of the language, justification of the design and the outline of the implementation has been presented in [2,4]. The details of the implementation of the EPL compiler for a single process specification has been documented in [tr2, tr3]. In real-time application with time-constraints and also in load balancing of parallel computation, the reliable estimates of the computational delays incurred in each participating process are of outmost importance. In [tr1] we presented a technique and a software tool for finding such estimates for programs specified in a definitional language (MODEL or EPL). Program integration problem is address in the EPL system by a configurator. The configurator implementation has been documented in [tr4], and reported in [9,8]. Particularly interesting environment for integration of real-time programs arises in ADA, and problems and solutions of this problem in the framework of definitional programming have been discussed in [6].

One of the new software tools developed for processing EPL specifications was conditional data dependence analyzer that was documented in [tr5] and reported in [5,7]. Our implementation of EPL code generator for shared memory Sequent Balance 21000 parallel computer employs mutual exclusion. We have developed a new, robust software solution to this problem, that requires small numbers of shared variables [8].

The annotations have been introduced in EPL as a means of a semanticly clean expression of user directives for partitioning and mapping of computation for parallel processing. The annotations are translated into purely functional EPL programs by the preprocessor documented in [tr6]. Finally, our initial results of investigation into

parallel fine-grain EPL scheduling has been presented in [3].

Technical Reports:

[tr1] M. Srinivasan, B. K. Szymanski, *A Timing Evaluator for C Programs Generated by the MODEL System, Department of Computer Science,* Technical Report 87-29, RPI, Troy, NY, 1987 (MS Thesis).

[tr2] D. E. Clark, *EPL - Equational Programming Language System Design - Intermediate Processing Steps,* Department of Computer Science, Technical Report 87-33, RPI, Troy, NY, 1987 (MS Thesis).

[tr3] B. Sinharoy, *EPL - Equational Programming Language, Parsing and Dimension Propagation,* Department of Computer Science, Technical Report 88-16, RPI, Troy, NY, 1988 (MS Thesis).

[tr4] K. Spier, *Propagation of Data Dependency through Distributed Cooperating Processes,* Department of Computer Science, Technical Report 88-24, RPI, Troy, NY, 1988 (MS Thesis).

[tr5] J. Bruno, *Analyzing Conditional Data Dependencies in an Equational Language Compiler,,* Department of Computer Science, Technical Report 90-4, RPI, Troy, NY, 1990 (Ph.D. Thesis).

[tr6] C. Ozturan, *Expressing User-Defined Parallelism in EPL (Equational Programming Language),* Department of Computer Science, Technical Report 90-29, RPI, Troy, NY, 1990 (MS Thesis).

Book chapters:

[1] N.S. Prywes and B. K. Szymanski, Software Development of Parallel Processing in a Distributed Computer Architecture, contributed to: *Supercomputing Systems*, L. Kryschev, and S. Kryschev (editors), Van Nostrand Reinhold, to appear in 1990.

[2] B. K. Szymanski, EPL - Parallel Programming with Recurrent Equations, contributed to *Parallel Functional Programming Languages and Compilers*, B. Szymanski (editor), ACM Press, to appear in 1991.

[3] B. Sinharoy, B. McKenney, and B. K. Szymanski, Scheduling EPL Programs for Parallel Processing, contributed to: *Languages, Compilers and Run-Time Environments*, J. Saltz (editor) Elsevier, to appear in 1991.

Articles in journals and conference proceedings:

[4] "Parallel Programming with Recurrent Equations," *International Journal on Supercomputer Applications* **1**, No. 2, pp. 44-74, 1987.

[5] J. Bruno and B. K. Szymanski, Use of Theorem Proving Techniques in Equational Language Compiler, *Proc. MCC-University Research Symposium*, Austin TX, MCC Press, 1987, pp. 173-182.

[6] B. K. Szymanski, Beyond ADA - Generating Ada Code from Equational Specifications, *Proc. of Sixth Annual National Conference on ADA Technology*, IEEE Press, Washington, D.C., 1988, pp. 494-499.

[7] J. Bruno, B. K. Szymanski, Conditional Data Dependence Analysis in an Equational Language Compiler, *Proc. Third International Conference on Supercomputing Systems*, ICS Press, Boston, MA, 1988, pp. 358-365.

[8] "B. K. Szymanski, A Simple Solution to Lamport's Concurrent Programming Problem with Linear Wait, *Proc. 1988 International Conference on Supercomputing*, St. Malo, France, ACM Press, 1988, pp. 621-626.

[9] K. Spier, and .B. K. Szymanski, Interprocess Analysis and Optimization in the Equational Language Compiler, *Proc. CONPAR90/VAPIV Conference* in Zurich, Switzerland, Lecture Notes in Computer Science, vol. 457, Springer Verlag, Bonn, 1990, pp. 324-335.